

REMARKS

Reconsideration and allowance of the subject application are respectfully requested.

The Examiner requests a clean copy of the reference "A Pipeline Push-Down Stack Computer," H. Stone, Chapter 12, 1969, pages 235-249. This document was listed in the IDS filed on March 31, 2004. Enclosed is a copy of this reference that has been reproduced from our original photocopy to be as clear as possible. Acknowledgement is requested.

Applicants appreciate the Examiner's withdrawal of the novelty rejection with regard to Yates (US 6,502,237). But the Examiner now rejects claims 1, 2, 7, 11-16 in view of a combination of Evoy (US 5,937,193) and Dickol (US 5,875,336).

Evoy discloses a computer system with a translator to translate platform-independent instructions into corresponding native instructions for execution by the processor. In addition to a software-based interpreter, one or more look-up tables are provided to map platform-independent instructions to native instructions. Contrary to the assertions in the Office Action, Evoy does not disclose the combination of features recited in claim 1.

Integer (v) of claim 1 recites that the hardware based execution logic includes scheduling support logic for "triggering a scheduling operation to be performed between program instructions for managing scheduling between threads or tasks irrespective of

whether a preceding program instruction was executed by said hardware based execution unit or said software based execution unit." Evoy lacks these claim features.

Dickol's computer translates non-native instructions to native instructions. The computer system comprises a processor, a memory and an instruction set converter. The processor can process only native instructions. The instruction set converter includes a semantics table and an information table. In response to an instruction fetch from the processor that relates to a non-native instruction stored in the memory, the instruction set converter translates the non-native instruction to a native instruction by accessing the semantics table and the information table (see Dickol col. 2, lines 32-46).

The Examiner contends that Dickol's col. 6, lines 6-19 discloses the claimed hardware execution unit with scheduling support logic. The cited passage describes how the instruction set converter 12 translates Java branch instructions (non-native instructions) to PowerPC (native instructions). In particular, an opcode from the semantics table of the instruction converter is combined with a branch offset value generated by a generate branch module 25 of the instruction set converter 12. The offset simply gives the number of bytes between a current instruction and the address of the instruction corresponding to the branch destination.

Generation of the branch offset signal in Dickol is not the same as generating "a scheduling signal for triggering a scheduling operation between program instructions...", as recited in claim 1, because the branch offset signal simply specifies the location of an instruction corresponding to a branch destination and does not trigger a scheduling

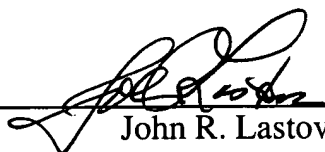
operation. Nor is the claimed scheduling operation simply execution of a next program instruction in a predetermined sequence, as implied by the Examiner. Claim 1 recites that the scheduling operation is (1) to be performed *between program instructions* and (2) for managing scheduling between threads or tasks. The latter feature (2) makes it clear that the claimed scheduling operation requires more than just executing the next translated program instruction. The specification, on page 2 lines 28-34, explains the scheduling problems associated with known systems (like Evoy's) where scheduling operations may be triggered part way through interpretation of a complex non-native program instruction.

Neither Evoy nor Dickol disclose or suggest the scheduling support logic of the hardware execution unit as specified by claim 1 integer (v). Similar language is found in claim 14. Accordingly, the application is in condition for allowance. An early notice to that effect is earnestly solicited.

Respectfully submitted,

NIXON & VANDERHYE P.C.

By: _____


John R. Lastova
Reg. No. 33,149

JRL:sd
Nixon & Vanderhye P.C.
901 North Glebe Road, 11th Floor
Arlington, VA 22203-1808
Telephone: (703) 816-4000
Facsimile: (703) 816-4100

CHAPTER 12

A PIPELINE PUSH-DOWN STACK COMPUTER

Harold S. Stone

*Stanford University
Palo Alto, California*

Abstract

This paper describes a new computer organization which is a hybrid of the pipeline computer and the push-down stack computer. The central characteristic of the computer organization is a hardware translator that performs dynamic register allocation. Because dynamic allocation can eliminate register conflicts that would otherwise exist, the new computer organization is potentially capable of higher performance than conventional pipeline computers. The new computer can support an unusual form of multiprocessing, and can eliminate processing delays that occur in pipeline computers when conditional branches are executed.

INTRODUCTION

An important technique for the design of high performance computers is the technique that is known as *pipeline design*. The essential characteristic of pipeline computers is that they can support concurrent operations in the following sense. An operation can be initiated within a module before the completion of the preceding operation. The rate of computation is determined by the rate at which operations can be initiated rather than by the time required to perform

The research reported on this project was sponsored in part by the Air Force Cambridge Research Laboratories, Office of Aerospace Research, Bedford, Massachusetts, under Contracts AF 19(628)-5520, and AF 19(628)-2902, and in part by the Atomic Energy Commission under contract AT(04-3)-326, P. A. 23.

This author is on leave from Stanford Research Institute.

736 PARALLEL PROCESSOR SYSTEMS, TECHNOLOGIES, AND APPLICATIONS

each operation. The IBM 360/91 and the Control Data 6600 are two examples of computers that make use of pipeline design.

Efficiency of pipeline computers depends strongly on the way that they are programmed. It is a nontrivial problem to write a machine language program that takes maximum advantage of the potential concurrency within a pipeline computer, and it is a far more difficult problem to construct a FORTRAN or ALGOL compiler that generates efficient programs for such a machine.^{1,2} In this paper, we describe a pipeline machine organization that can be programmed very easily to achieve high efficiency computation. The computational portions of the processor are assumed to be similar to conventional pipeline computers. High efficiency is achieved by deferring part of the compilation process until execution. Instructions are coded in a polish string format, but as each instruction reaches the execution unit, it is recoded into a 3-address format. A translator module performs the recoding in such a way as to maintain high utilization of processor resources. Since register allocation is effectively performed dynamically during execution rather than fixed prior to execution, greater concurrency is possible than with a fixed allocation.

The second section of this paper, "A Model of a Pipeline Computer," gives a brief outline of the structure of a pipeline computer. The third section describes the structure of the pipeline push-down stack computer, including a description of the translation algorithm, and the fourth section contains a brief discussion of characteristics of the computer organization that support multiprocessing, executive control, and efficient memory access. A brief summary appears at the end of the paper.

A MODEL OF A PIPELINE COMPUTER

In the current computer vernacular, the term "pipeline" refers to the ability to execute two or more operations concurrently within a module when the interval between initiation of operations is smaller than the process time associated with each operation. A module that operates in pipeline mode can be executing one operation in an early process phase at the same time that it is executing another operation in a late process phase. The term *pipeline* is particularly descriptive because the rate of flow in a petroleum pipeline depends on the rate at which material is pumped into the pipeline, and does not depend directly on the time required to transfer material.

The particular type of pipeline operation described in this paper is a mode of execution of instructions in a central processor.

The computer model described below permits instructions to be issued from an execution station at a rate that is higher than the rate possible in a conventional computer. Several instructions may simultaneously be in process in this computer. The model is intended to exemplify the behavior of computers like the CDC 6600 and the IBM 360/91, but no attempt is made to characterize the actual physical design of such computers.

Figure 1 shows the pipeline computer organization. The processing portion of the computer is contained in a collection of registers, function units, and in a switching network that connects function units to registers. Each functional unit can perform an operation on one or two operands and return a single result. Operations may be fixed-point or floating-point arithmetic operations such as ADD, SUBTRACT, MULTIPLY, and DIVIDE; logical operations such as AND, OR, and EXCLUSIVE OR; magnitude comparisons such as GREATER THAN, EQUAL, and LESS THAN; field operations such as SHIFT, ISOLATE, and NORMALIZE; and other miscellaneous operations. We do not specify the operations of the function units themselves for there is considerable freedom in this choice. Each function unit might be capable of performing only one operation, any operation within a class of related operations, or any operation in the processor repertoire. Within the collection of function units, each function unit may be unique, may be one of a subset of identical units, or all function units may be identical. There is assumed to be at least one function unit that can perform each operation. Otherwise the capabilities of the individual units is a design parameter that can be specified on a cost-performance basis.

We assume that instructions specify one or two operand registers, an operation, and a result register. For example, typical instructions are $R_1 \leftarrow R_2 + R_3$, and $R_1 \leftarrow R_6/R_5$. Memory fetches and stores are coded in the format $R_1 \leftarrow X$ and $Y \leftarrow R_4$, respectively, where we use X and Y to represent memory addresses.

In addition to the processing hardware, the computer contains a unit called the *precedence memory*, an execution station, and an instruction queue. Instructions are processed in the following manner. Instructions are removed from the instruction queue one-at-a-time at the clock rate of the computer. The execution station passes each instruction to the precedence memory if it is not a conditional branch. If the execution station cannot identify the next instruction because of an uncomputed

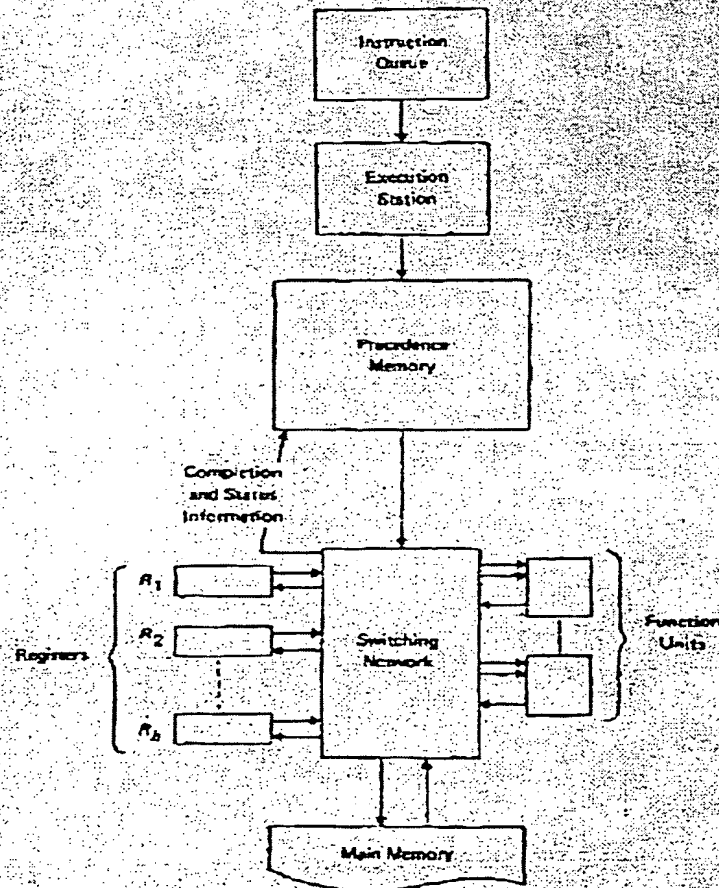


Fig. 1. A Model of a Pipeline Coprocessor.

branching condition, the execution station may pursue one of several courses of action in anticipation of the outcome of the conditional test. The precise strategy followed in this case is not an essential feature of the model.

The precedence memory is an active memory that defers or initiates instructions according to the following algorithm. As each instruction enters the precedence memory, the instruction is initiated if it can be

A PIPELINE PUSH-DOWN STACK COMPUTER 239

initiated, otherwise it is deferred. Deferral of instructions can be caused by unavailability of one or both operands, function unit, or result register. With each deferred instruction the precedence memory records the status of items required for the instruction. Status information is updated as each instruction that has been initiated reaches completion. Status changes that permit any deferred instruction to be initiated are detected in the precedence memory, thereby causing initiation of such instructions. Information pertaining to completed instructions is purged from the memory to permit new instructions to be entered.

To illustrate the operation of the pipeline computer, consider the steps in the evaluation of the numerical assignment $A = B \times C + D \times E$. A typical instruction sequence for this calculation is:

- 1) $R_1 = B$
- 2) $R_2 = C$
- 3) $R_3 = B \times C$
- 4) $R_4 = D$
- 5) $R_5 = E$
- 6) $R_6 = R_4 \times R_5$
- 7) $R_7 = R_6 + R_3$
- 8) $A = R_7$

All four memory fetches can be initiated concurrently because the instructions use different result registers. Instructions (3) and (6) can also be executed concurrently, but each of these instructions can be initiated only after both operands have been fetched from memory. Figure 2 shows the extent of concurrent execution for this example that might occur within a typical pipeline computer.

Further discussion of the characteristics and organization of pipeline computers can be found in Thornton, Chen, and Anderson *et al.*

A PIPELINE PUSH-DOWN STACK COMPUTER

The model of a pipeline computer described in the previous section can achieve a high degree of concurrency during execution of instructions, but the degree of concurrency is sensitive to register assignments in the following sense. Two instructions that reference a common register cannot be executed concurrently if one uses the register for an operand and the other uses the register for a result. Hence, register allocation imposes sequential constraints on the execution of

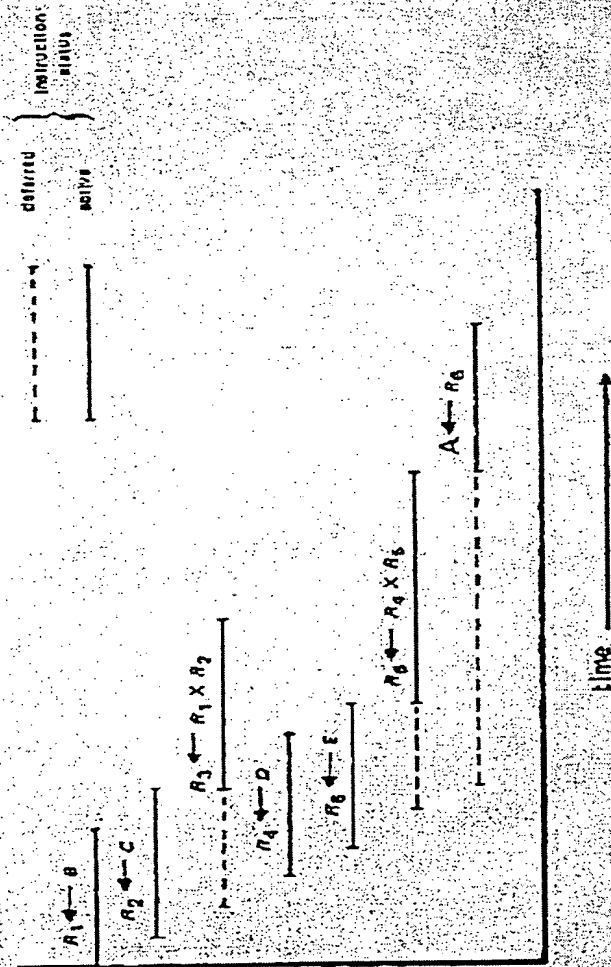


Fig. 2. Concurrent Execution of Instructions in a Pipeline Computer

instructions. "Good" register assignments are those which impose only the essential sequential constraints, and no more. In today's pipeline computers, register allocation is part of the compilation process, but it is the thesis of this paper that register allocation should occur dynamically. Register allocation that is fixed for an entire execution of a program is almost certainly suboptimal, because many segments of programs can be entered in different contexts, and a different optimal register assignment for the segment may exist for each context. The remainder of this section is devoted to the description of a machine organization in which register allocation occurs dynamically.

A machine organization which performs dynamic register allocation is shown in Fig. 3. We have added three modules to the pipeline computer. The module identified as a *translator* accepts instructions in a push-down stack format that will be described shortly. The output of the translator is a stream of instructions in the standard pipeline computer format. The "idle register queue" is an active memory that maintains a list of unassigned registers. The translator uses this list together with information in the "push-down stack" as it performs the task of register allocation. The salient feature of this machine organization is that register references in the input instruction stream are implicit rather than explicit. Explicit register references are compiled in instructions just prior to their actual execution, and the assignments are made on the basis of immediate availability of registers. Therefore, no instruction need be deferred unnecessarily because of register assignment conflicts.

The instructions at the input of the translator are in push-down stack format. That is, they operate on the top register or top two registers of a push-down stack. The instructions VALUE X and NAME X place the value and the memory address of symbolic variable X at the top of the stack, "pushing down" the previous contents of the stack. Arithmetic instructions ADD, SUB, MUL, and DIV use the top two registers of the stack as operands, compute an arithmetic result, and replace the top two registers with a single result. Arithmetic operations effectively "pop-up" the stack one time. The instruction STORE interprets the top element of the stack as a value and the next to top of the stack as an address. STORE causes the value at the top of the stack to be placed in the memory address given by the next to top element of the stack, and deletes both of the top two items from the stack. A complete instruction repertoire necessarily includes instructions for indexing, logical operations, conditional and unconditional branching, and subroutine calls. For the purposes of this paper, the seven instructions

242 PARALLEL PROCESSOR SYSTEMS, TECHNOLOGIES, AND APPLICATIONS

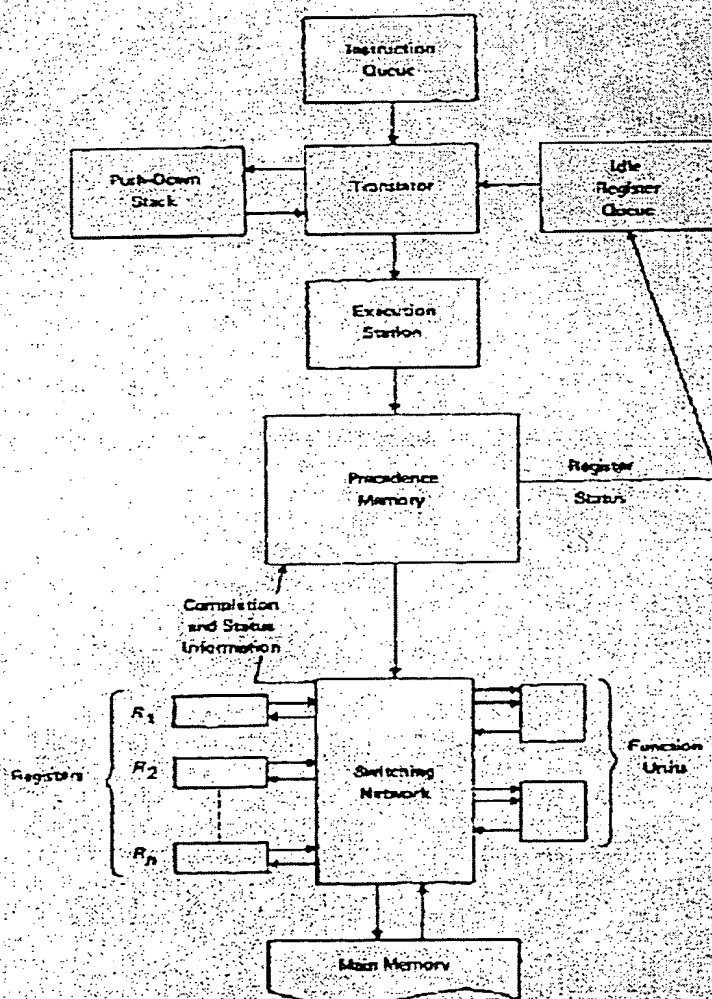


Fig. 3. A Model of a Pipeline Push-down Stack Computer

that have been described thus far are sufficient to illustrate the concept of the hardware translator. A complete description of a computer that uses push-down stack instructions can be found in Burroughs.⁶

A PIPELINE PUSH-DOWN STACK COMPUTER 243

The calculation $A = B * C + D * E$ given in the previous section can be written as a program for a push-down stack computer as follows:

- 1) NAME A
- 2) VALUE B
- 3) VALUE C
- 4) MUL
- 5) VALUE D
- 6) VALUE E
- 7) MUL
- 8) ADD
- 9) STORE

A characteristic of the stack instruction set that is exemplified in the program above is that the program contains no explicit references to registers in a central processor other than the implicit references to the registers at the top of a push-down stack. When these instructions are executed in a push-down stack machine, the *implicit* references are turned into *explicit* references. It is not unusual, therefore, in stack organized computers to find that a particular instruction might refer to different machine registers at different execution times. Since register references in stack organized computers are not bound until the actual execution of instructions, instructions in push-down stack format are ideal for supporting dynamic register allocation.

Algorithms for translating a push-down stack format to a standard register format are not new. They have been incorporated in nearly all compilers. Compilers generally translate source text into an intermediate text known as a "polish string," which is roughly equivalent to a program in push-down stack format. The intermediate text is then translated into standard format by assigning registers to the instructions. See, for example, Horowitz *et al.*¹ For our model of a pipeline computer, we shall state an algorithm for translating the seven instructions VALUE, NAME, ADD, SUB, MUL, DIV, and STORE into a 3-address format. The algorithm can be generalized to a full stack instruction repertoire in a straightforward manner.

In the processor shown in Fig. 3, the idle register queue contains indices of unassigned registers. If the queue is empty when a request for a register is issued by the translator, the queue defers response until an idle register becomes available. (If a program issues a sufficiently long sequence of VALUE instructions, all registers can become unavailable simultaneously without a possibility of a register becoming available. We assume that programs never require more registers to be active

pt
at

244 PARALLEL PROCESSOR SYSTEMS, TECHNOLOGIES, AND APPLICATIONS

simultaneously than the number of registers in the processor. The assumption is valid, in practice, for there exist several mechanisms for enforcing the assumption.)

Each cell of the push-down stack can contain a register index or a memory address. The stack itself is under control of the translator. The translation algorithm for our seven instruction repertoire is the following:

- 1) VALUE: The translator obtains a register from the idle register queue. Let this register be R_i . R_i is stacked on the push-down stack and the instruction VALUE X is translated to $R_i \leftarrow X$.
- 2) NAME: The instruction NAME X causes the address of X to be stacked. No translated instruction is produced.
- 3) ADD, SUB, MUL, DIV: Let the top two entries of the stack be the registers R_i and R_j . An idle register is obtained from the idle register queue. Let this register be R_k . Then the instruction $R_k \leftarrow R_i \langle \text{op} \rangle R_j$ is produced where $\langle \text{op} \rangle$ is $+$, $-$, \times , or $/$. R_i and R_j are popped-up and R_k is pushed-down.
- 4) STORE: Let the top entry of the stack be the register R_i and the next to top entry be the memory address X . Then the translator produces the instruction $X \leftarrow R_i$ and the stack is popped-up twice.

To illustrate the algorithm, assume that the idle register queue initially holds the indices 1 through 7 in ascending order. Then if the translator operates on the program for the calculation of $A \leftarrow B \times C + D \times E$ as given in this section in push-down stack format, the output of the translator will be the program for the same calculation in 3-address format that appears in the previous section.

The machine organization in Fig. 3 is a *hybrid* in the sense that it has characteristics of both the push-down stack computer and the pipeline computer. Since the translation process can operate concurrently with execution, and since register allocation is guaranteed to be optimal or near optimal, the new computer organization is potentially capable of higher performance than the pipeline computer. The new organization can also achieve higher performance than that obtainable in a conventional push-down stack computer. The latter claim is true because operations on the top of the stack impose sequential constraints on the execution of instructions in a push-down computer. It is not possible, in general, to execute push-down instructions concurrently without translation since the context of each instruction depends on the stack manipulation of the immediately preceding

instructions. Any computer organization that can support concurrent execution of push-down stack instructions must simulate stack manipulation in some manner, and consequently must perform a process similar or equivalent to the translation process in the new computer organization.

There are several interesting facets of the new processor organization that deserve separate attention. We treat each of these in the next section. Further information on the organization and design of this pipeline push-down stack computer appears in Elspas *et al.*²

CHARACTERISTICS OF THE PIPELINE PUSH-DOWN STACK COMPUTER

In this section we describe how several conventional and advanced techniques of computer design are compatible with the organization of the pipeline push-down stack computer.

Multiprocessing

A multiprocessor can be constructed in a conventional way by replicating copies of the processor shown in Fig. 3. However, an unusual form of multiprocessing can also be supported by a single processor. Several independent instruction streams can share a single processor provided that each of the instruction streams has its own program counter, translator module, and push-down stack module. The number of registers and functional units must be chosen to satisfy the processing requirements of multiple instruction streams.

There is a potential advantage in operating one processor with multiple instruction streams instead of using multiple copies of a processor. Resources that are not in use temporarily by one process can be put to use by another process so that the multiple instruction stream processor can achieve higher utilization of its resources than a single instruction stream processor. If high resource utilization more than balances the performance degradation due to increased size and complexity, the multiple instruction stream processor is a potentially better vehicle for multiprocessing than a computer with multiple processors.

Execution of High Priority Processes

In many situations, high-priority processes can preempt a processor for brief periods of time. These processes may be associated with executive control, interrupt processing, or control of the external environment under a real-time deadline. In the conventional computer, a high-priority process either preempts an entire processor, or uses hardware that is private to the process. In the new computer organization, a high-priority process can share a processor with one or more low-priority processes.

A high-priority process can gain access to a processor as an independent instruction stream. To guarantee that the high-priority process obtains all of the registers and functional units that it requires, we postulate that a high-priority process can reserve resources for a short period of time. Low-priority processes can continue to operate, but with a smaller pool of registers and function units than would otherwise be available. Since the precedence memory and the idle register queue maintain a table of all available and unavailable resources, resources can be reserved or freed by making appropriate status changes in the resource table.

Look-aside Memory

The precedence memory can be used as a look-aside memory to eliminate unnecessary main memory references. Suppose, for example, that the instruction $X - R_1$ has been deferred or is in the process of execution when the instruction $R_2 - X$ enters the precedence memory. The precedence memory must be able to detect the fact that both instructions refer to the same memory address, X , and that the second instruction might have to be deferred until the completion of the first instruction. The precedence memory can eliminate a reference to main memory by changing the second instruction to the form $R_2 - R_1$, since register R_1 has a copy of the contents of X . In this form, the instruction can be executed concurrently with the instruction $X - R_1$.

Conditional Branches

A novel method exists for avoiding delays that are normally associated with conditional branches. The processor can proceed along both branches of a conditional branch until the correct branch is identified.

A PIPELINE PUSH-DOWN STACK COMPUTER 247

The following method is used to guarantee that computations on the two branches of a conditional branch cannot conflict with each other.

Computation proceeds normally until a conditional branch is reached. At that point, a second translator and push-down stack are brought into operation. The two translators then proceed concurrently along opposite paths of the conditional branch. Since register assignments are dynamic, the computations on the separate paths are guaranteed to use different sets of registers. Instructions generated by each translator must be identified with their respective translators so that all instructions associated with the incorrect branch can be purged from the precedence memory when the branching condition computation is completed. If either translator reaches a STORE instruction, it halts until the correct branch is identified since we cannot allow incorrect branches to modify main memory.

Translation into Two-address Instructions

In the pipeline processor, it may be advantageous to use a two-address format instead of the three-address format that has been assumed. Two-address instructions take the form $R_i - R_j \langle op \rangle R_j$, where the result register is the same as the first operand register. Two-address instructions impose a smaller storage requirement on the precedence memory than three-address instructions, and are also somewhat simpler to translate. In two-address processors, when a translator encounters the instruction ADD in the instruction stream, it can issue the translated version $R_i - R_j + R_j$ without accessing the idle register queue. The indices i and j are assumed to be the top two indices on the push-down stack, and the stack is popped-up once. No new index need be rewritten in the stack. Translation into two-address format requires access to the idle register queue and requires that the contents of one register in the stack be modified.

Compiler Methods for Maximizing Parallelism

In spite of the fact that compilers need not perform register allocation for this computer organization, compilers can produce instruction sequences that take into account the parallelism available in the new processor organization. For example, the computation $A = B + C + D + E$ can be calculated as if it were $A = ((B + C) + D) + E$, or as if it were $A = (B + C) + (D + E)$. The latter is clearly more suitable for a pipeline

248 PARALLEL PROCESSOR SYSTEMS, TECHNOLOGIES, AND APPLICATIONS

computer because the terms $B + C$ and $D + E$ can be calculated concurrently. Algorithms for maximizing concurrency have been studied by several authors—Squire, Hellerman, Stone, and Baer *et al.*^{9,10,11,12} The one-pass algorithm given by Stone is specifically designed for this machine organization, while the other algorithms are designed for more conventional pipeline computers.

Although there are several other minor points that bear discussion, the observations of this section cover the major characteristics of the new computer organization. One point that has purposely been avoided has been the organization of main memory. It should be clear that the processor can be used with any advanced memory organization of sufficient size and speed.

SUMMARY

The processor organization described in this paper is a hybrid of the push-down stack computer and the pipeline computer. Because it uses dynamic register allocation to minimize register conflicts, it is potentially capable of higher performance than that attainable by today's pipeline computers. The distinguishing characteristic of the new computer organization is a hardware translator that performs register allocation at execution time with the aid of algorithms that heretofore have been found in compilers.

A fundamental concept that underlies the processor organization reported here is that resource allocation of a general nature is inherently a dynamic process. In order to attain the highest achievable performance, we must expect to use hardware to perform dynamic resource allocation. Several questions pertaining to scheduling, memory allocation, and task partitioning are still open in this area, and must be resolved if we are to overcome the present limitations of high-speed computers.

References

1. R. W. Allard, K. A. Wolf, and R. A. Zemlin, "Some Effects of the 6600 Computer on Language Structures," *CACM*, 7: No. 2 (February, 1964), 112-19.
2. H. L. Nelson, "Program Optimizing Techniques for the CDC 6600 Central Processor," UCRL-12489 (University of Calif. Lawrence Rad. Lab. [1965]), also NASA N66-20536.

A PIPELINE PUSH-DOWN STACK COMPUTER 249

3. J. E. Thornton, "Parallel Operation in the Control Data 6600," *AFIPS FJCC Conf. Proc.*, 26: Part II (Washington, D. C.: Spartan Books, 1964), 33-40.
4. T. C. Chen, "The Overlap Design of the IBM System/360 Model 92 Central Processing Unit," *AFIPS FJCC Conf. Proc.*, 26: Part II (Washington, D. C.: Spartan Books, 1964), 75-80.
5. D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM J. R. & D.*, 11: No. 1 (January, 1967), 8-24.
6. *Burroughs B5500 Information Processing Systems Reference Manual* (Burroughs Corp., [Detroit, Mich., 1964]).
7. L. P. Horwitz *et al.*, "Index Register Allocation," *JACM*, 13: No. 1 (January, 1966), 43-61.
8. B. Elspas *et al.*, "Investigation of Propagation-limited Computer Network," AD-637-769 (Phase III Final Report, S.R.I. Project 4523, Air Force Cambridge Research Laboratories, Office of Aerospace Research [Bedford, Mass., June, 1966]).
9. J. S. Squire, "A Translation Algorithm for a Multiple Processor Computer," *Proc. 18th ACM Nat. Conf.* (Denver, Colorado, 1963).
10. H. Hellerman, "Parallel Processing of Algebraic Expressions," *IEEEEC*, EC-15: No. 1 (February, 1966), 82-90.
11. H. S. Stone, "One-pass Compilation of Arithmetic Expressions for a Parallel Processor," *CACM*, 10: No. 4 (April, 1967), 220-23.
12. J. L. Baer and D. P. Bovet, "Compilation of Arithmetic Expressions for Parallel Computations," *IFIPS Congress 68*, August, 1968, pp. B4-B10.

6600
364).

ntial
55)).

**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ **BLACK BORDERS**
- ☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☐ **FADED TEXT OR DRAWING**
- ☐ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☐ **SKEWED/SLANTED IMAGES**
- ☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☐ **GRAY SCALE DOCUMENTS**
- ☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER:** _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.